1900

PASCAL USER'S GUIDE

VERSION 2
AUGUST 1977

Department of Computer Science

The Queen's University of Belfast
Belfast BT7 1NN Northern Ireland Telex 74487

F. J. Smith, M.R.I.A.,
Professor of Computer Science

Tel. 45133 Ext. 3220/1

22 February 1978

Dear Mr. A. W. Moore,

Please find enclosed a magnetic tape containing the 1900 PASCAL system, mark 2 - issue 3. We apologise for the delay in sending the tape. The associated documentation consists of:

(1) Tape specification

(2) System specification

(3) Users Guide

(4) An insertion to the Users Guide describing the source library enhancement.

It should be noted that there are significant operational differences between this issue and previous issues of mark 2, and of course mark 1. Please read the system specification carefully. The tape now includes a George 3/4 macro (developed at the University of Glasgow) for running PASCAL jobs. This may help in installing the new system.

Also note that the binary compiler (and binary postmortem generator) on the tape were generated with checks on. By recompiling the associated source subfiles with checks off, their size will be reduced (by approx. 3% for the compiler) and their speed improved (by about 8% for the compiler).

Yours sincerely,

Kathleen M. McConnell

p.p. J. WELSH.

EN

PASCAL SYSTEM TAPE : PROGRAM DIST. (31)

This tape originated on an ICL 1906S machine under the GEORGE 4 operating system. Its physical characteristics are

either 9 track: (phase encoded, 1600 bpi, odd parity, written on 2505 tape system)

or 7 track: (NRZI,556 bpi, odd parity, written on 1971 tape system).

It was created using the ICL utilities #XKYA and #XPMV. It contains

(1) A search program #DIST generated by #XPMV

(2) Five source subfiles created by #XKYA which are

(i)	COMPILER	-	the PASCAL source of the full 1900 PASCAL compiler	15810
(ii)	MONITOR	-	the PLAN source of the run-time support package used by the full PASCAL system	5779
(iii)	POST-GEN	-	the PASCAL source of the post-mortem generator	2416
(iv)	LIBEDIT	-	GEORGE 3/4 editor instructions to add the source library enhancement to the COMPILER source	551
(v)	MACRO	-	a GEORGE 3/4 macro to run PASCAL programs	251

(3) A subfile LIBRARY FILE, created by #XPMV, which contains two binary programs:

(i)	#PASQ	-	the 1900 PASCAL compiler
(ii)	#POST	-	the postmortem generator

The source subfiles may be extracted using #XKYA. The binary programs may be loaded by directives

```
e.g. FI   #PASQ   #DIST

     FI   #PØST   #DIST
```

or any equivalent sequences. How to run the systems is described in separate documentation.

If any one wishes to amend or extend the systems, the necessary procedures are discernible from the source listings. However it may be advisable to consult the authors before doing so.

J. WELSH.

Department of Computer Science
Queen's University
Belfast BT7 1NN.

1900 PASCAL User's Guide.

This document describes the PASCAL programming language as implemented on ICL 1900 series computers. The language implemented is essentially that defined by Jensen & Wirth's 'Pascal User Manual and Report' and this document is intended only as a supplement to that text, in effect replacing Chapters 13 and 14 thereof.

The Guide consists of three sections. Section 1 defines the version of PASCAL implemented, Section 2 outlines the facilities provided by the 1900 PASCAL system, and Section 3 explains how to run 1900 PASCAL programs at a particular computing installation, in this case the Computer Centre of Queen's University.

The 1900 PASCAL compiler was written at the Queen's University Belfast by J. Welsh, C. Quinn and K. McShane. The diagnostic facilities and compiler directives were added at the University of Glasgow by D.A. Watt and W. Findlay. This Guide is a collation of material written in Belfast and Glasgow. Any comments or queries on its content should be directed to the address below.

J. Welsh.

Department of Computer Science
Queen's University
Belfast.

Version 2

August 1977.

Chapter 1.

1. The 1900 Pascal Language

The dialect of Standard Pascal implemented on ICL 1900 computers is that defined in [Jensen and Wirth, "PASCAL User Manual and Report"] subject to the following amendments.

1.1 Vocabulary

Only the capital letters are available in programs or data. Symbols written underlined in the Report, the word-delimiters, are written in 1900 Standard Pascal without underlining and without any surrounding escape characters.

The characters { and } are not available, so comment brackets are written as (* and *) instead. The symbol # is accepted as an alternative for <> and @ is accepted as an alternative for ↑. Blanks, end-of-lines and comments are considered to be separators and may be inserted anywhere except within word-delimiters, identifiers, numbers and the symbols :=, .., <=, >=, <>, (*, *) .

1.2 Identifiers and numbers

Only the first 8 letters and digits of an identifier are significant. Identifiers which do not differ in the first 8 characters are considered to be identical. Word-delimiters are reserved and must not be used as identifiers. At least one separator must appear between adjacent word-delimiters, identifiers or numbers.

Integer constants may be written in either decimal or octal notation, according to the following syntax:

```
<unsigned integer> ::= <digit sequence>|<octal digit sequence>
<octal digit sequence> ::= <octal digit>{<octal digit>}B
<octal digit> ::= 0|1|2|3|4|5|6|7
```

1.3 Data types

The type INTEGER is given by the definition:-

```
    TYPE INTEGER = -MAXINT .. +MAXINT
```

where MAXINT=8388607 on the ICL 1900.

The type REAL is defined by the ICL 1900 floating point representation which allows values in the approximate range $\pm 1.0E\pm 76$ with about 11 significant figures. Arithmetic operations on REAL values imply rounding.

The type CHAR is defined by the ICL 1900 six-bit internal character set. The ordering of the type is thus:-

0	1	2	3	4	5	6	7
8	9	:	;	<	=	>	?
ƀ	!	"	#	£	%	&	'
(	)	*	+	,	-	.	/
@	A	B	C	D	E	F	G
H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W
X	Y	Z	[	$	]	↑	←

SET types may be defined over any base type which is:-

(a) an enumerated type with not more than 48 values, or

(b) a subrange of type INTEGER with a minimum element >=0 and a maximum element <48, or

(c) a subrange of any other ordinal type which does not include any value beyond the 48th value in the ordering.

A restriction imposed by the compiler is that the empty set (i.e. []) cannot be given as an actual parameter in a call of a formal procedure.

In RECORD types with variant parts explicit tagfields must be specified. (This is in accordance with Wirth's "second thoughts" on the subject.)

FILES cannot be assigned, cannot be passed as value parameters, and cannot occur as components of any other structured type.

An additional predefined type ALFA is available, with the definition:-

```
TYPE ALFA = PACKED ARRAY [1..8] OF CHAR
```

1.4 The standard procedure WRITE

If no minimum field length parameter is specified, the following default values are assumed:-

Type	Default
INTEGER	8
REAL	16 (and the exponent is always given in the form E±dd)
BOOLEAN	5
CHAR	1
a string	the length of the string

The end of each line in a textfile f must be explicitly indicated by WRITELN(f), where WRITELN(OUTPUT) may be expressed simply as WRITELN. If a textfile is sent to a lineprinter, no line may contain more than 120 characters.

1.5 The program heading

The program heading serves to identify the FILE variables through which the program communicates with its environment. It has the form:-

```
<program heading> ::= PROGRAM <identifier> ( <permanent files> ) ;

<permanent files> ::= <identifier> {, <identifier> }
```

The program identifier must be at least four characters long.

Each permanent file identifier must also be declared as a FILE variable in the main program block. However the standard FILEs INPUT and OUTPUT must not be declared, but do have to be listed in the program heading if they are used. For those programs which use only the standard FILEs:-

(a) a reduced form of program heading may be used, viz.:-

```
PROGRAM <identifier> ;
```

which is equivalent to:-

```
PROGRAM <identifier> (INPUT, OUTPUT) ;
```

(b) the program heading may be omitted entirely, in which case the compiler assumes a heading of the form:-

```
PROGRAM PASCAL (INPUT, OUTPUT) ;
```

1.6 FILE variables and external files

Each FILE variable is implemented using an appropriate I/O device, as follows:-

(a) the standard permanent file INPUT is implemented as a card reader with unit number zero (*CR0)

(b) the standard permanent file OUTPUT is implemented as a line printer with unit number zero (*LP0)

(c) all other permanent files are implemented as direct access files with unit numbers 0, 1, 2 ... allocated in the order of their occurrence in the program heading. The files must already have been created with filenames which are the first 8 characters of the corresponding file variable identifiers. They will be opened on entry to the program in a mode which permits reading and writing.

(d) all non-permanent FILE variables are implemented as direct-access scratch files, created on entry to the block in which they are declared and erased on exit.

For files represented on read/write storage media (direct access, magnetic tape) the procedure RESET must be applied before reading takes place and REWRITE before writing takes place. For files represented on basic peripheral devices (card readers, line printers) these procedures need not be called.

The format used in recording files on magnetic tape and direct-access media is peculiar to the Pascal system. It is not designed to be compatible with other 1900 Series software. Textfiles which are to be processed by non-Pascal programs (such as an Editor) must therefore be manipulated as card reader and line printer files.

An input card reader file is assumed to be terminated by the occurrence of a line with asterisks in columns 1 to 4, in accordance with the ICL convention.

Note that the external file is opened on entry to the block in which the file variable is declared. Thus an external file must be provided for each file variable, regardless of whether the program actually performs any explicit data transfers upon it or not. If a file variable is not to be accessed in a run, a suitable empty external file may be associated with it to satisfy this rule.

Details of how the default device types, unit numbers and opening modes may be changed and of how actual files may be associated with them will be found in chapter 3.

1.7 Predefined functions and procedures

The predefined functions and procedures cannot be passed as actual parameters.

The following additional predefined routines are available ;

Procedures

DATE(a) assigns to the ALFA variable a the current date, expressed in the form dd/mm/yy.

TIME(a) assigns to the ALFA variable a the current time of day, expressed in the form hh/mm/ss.

MILL(i) assigns to the INTEGER variable i the CPU (mill) time used by the program so far, expressed in milliseconds. Since this time may include time spent before the execution of the main program proper commences, accurate program timings can be made only by taking the difference of successive values returned by MILL. The value given to i will be significant only if the machine is fitted with a mill-timer.

HALT(x) terminates execution of the program, displaying the string value x. Permanent files attached to the program are closed in the process.

Functions

CARD(s) yields the cardinality (i.e. the number of members) of the SET value s.

The correct execution of programs which include functions with side-effects is not guaranteed in 1900 Pascal.

1.8 Initialising global variables

A non-standard facility enables the value of global variables (i.e. those declared in the outermost block) to be initialised at compile-time. In some programs this may save writing many assignment statements which are obeyed only once. The initialisation takes the form of a value part which may immediately follow the variable declaration part of the program block. This value part has the form:-

```
<value part> ::= <empty>
          | VALUE <value specification>{;<value specification>};
<value specification> ::= <identifier> = <initial value>
<initial value> ::= <constant>
          | ( <constant>{,<constant>})
```

Note the following:-

(a) The symbol VALUE is a reserved word.

(b) Not all variables declared need be initialised, but those which are must appear in the value part in the same order as they appear in their declarations.

(c) The initial value specified must be identical in type with the variable being initialised.

(d) A constant list may be used to initialise array or record variables, each component value being specified in the order of its occurrence in the structured value. Components of multi-dimensional arrays are ordered in a row-major fashion.

(e) A packed array of characters may be initialised with a string. Other packed types may not be initialised.

1.9 Imbedding 1900 machine code instructions in PASCAL

Although PASCAL provides a powerful and flexible means of expressing a wide range of programming operations it is sometimes necessary or convenient to resort to machine level e.g. to control some peripheral device directly. For this reason the 1900 PASCAL compiler permits the imbedding of machine code instructions in PASCAL programs, by means of a built-in procedure ICL and a built-in function ADDRESSOF described below.

From a PASCAL viewpoint it is convenient first to classify 1900 machine instructions as follows

(i) branch instructions involving 3 fields in the general form

```
F  X  N
```

where N is a program label or address, e.g.

```
BZE  5  L4
```

(ii) storage addressing instructions involving 3 or 4 fields in the general form

```
F  X  N(M)
```

where N is the name of a storage location and M may be omitted implying no modification e.g.

```
ADX  5  JOHN
STO  6  FRED(2)
```

(iii) absolute instructions involving 3 or 4 fields in the general form

```
F  X  N(M)
```

where N is an absolute numeric value and again M may be omitted, e.g.

```
LDN  4  100
ADX  6  5
LDCT 2  0(2)
```

1.9.1 The built-in procedure ICL

The generation of instructions as classified above within a 1900 PASCAL program is provided by a built-in procedure ICL which accepts either 3 or 4 parameters in the general form

```
    ICL  (F, X, N) ;
or  ICL  (F, X, N, M) ;
```

The effect of each appearance of ICL in the source PASCAL program is to generate in the corresponding object program the machine instruction defined by its particular parameters F,X,N,M. These must obey the following rules:

(1) <u>The function parameter F</u>

In all cases F must be an integer constant which is the function code of the required machine instruction. Since PASCAL allows octal constants this value can be copied directly from the ICL literature, e.g.

```
ICL (050B, .... ) ;
```

Alternatively the user may first define the PLAN instruction mnemonics as PASCAL symbolic constants e.g.

```
CONST  LDX = 000B ;  BZE = 050B ; ....
```

and the use of these mnemonics in his calls of ICL e.g.

```
ICL (BZE, .... ) ;
```

(2) <u>The accumulator parameter X</u>

In all cases X must be an integer constant in the range 0 - 7, e.g.

```
ICL (BZE, 5, ...) ;
```

For those instructions in which the accumulator field is not significant, e.g. STOZ, X must be specified as zero e.g.

```
ICL (STOZ, 0, ....) ;
```

(3) <u>The operand parameters N,M</u>

The form which the N,M parameters may take depends on the class of instruction being generated, as follows

(i) for <u>branch instructions</u> N must be a PASCAL label defined in the same block, and the M parameter must not appear, e.g.

```
  ICL (BZE,5,3) ;
  ICL (ADX,4,6,2) ;
3:ICL (SBX,7,4) ;
```

Branches to absolute locations or to any point other than a labelled PASCAL statement are not permitted.

(ii) for storage-addressing instructions N must be a PASCAL program variable which is

(a) simple, involving no subscripting ([]), field selection (.), or indirection (↑)

and (b) either local, i.e., declared in the current block, or global, i.e., declared in the main program, and the M parameter must not appear, e.g.

```
ICL (LDX, 6, I) ;
```

where I is a local or global variable.

A global variable addressed by procedure ICL must also be in Lower Data, i.e. have an address less than 4096. A local variable addressed by procedure ICL must be in the first 4096 words of local storage for the block in which it is declared.

In general addressing PASCAL variables, even those obeying rules (a) and (b) above, may involve modification and the PASCAL compiler will automatically insert the appropriate modifier field in the generated instruction. The user must neither attempt to specify an additional modification by supplying an M parameter, nor make any assumption about the modification used.

(iii) for absolute instructions N must be an integer constant in the range 0 - 4095 and M if it appears must be an integer constant in the range 0 - 3.

Since machine code instructions generated by the ICL procedure are in general interspersed among instructions generated from source PASCAL statements by the PASCAL compiler it is necessary to impose certain additional constraints on the use of ICL to ensure that the two types of instruction do not interfere with one another.

(i) The contents of accumulator X1 are significant throughout machine code generated by the PASCAL compiler. The procedure ICL must never be used to generate a machine code instruction which, directly or indirectly, may alter the contents of X1.

(ii) Where the execution of a machine code sequence generated from a source PASCAL statement or statements occurs between the execution of two ICL generated instructions, no assumptions may be made about the contents of the accumulators when the second ICL-generated instruction is reached. <u>In general the execution of code generated from a source PASCAL statement may alter the contents of any or all of the accumulators</u>.

1.9.2 The built-in function ADDRESSOF.

The restrictions (a), (b) imposed on the N field of storage addressing instructions generated by the built-in procedure ICL make it impossible to generate machine code instructions addressing certain types of variable by this procedure alone. To enable the addressing of these variables 1900 PASCAL provides a built-in function ADDRESSOF which, given any valid variable as parameter, produces as result an integer which is the absolute address of the current instance of that variable. This allows the indirect addressing of e.g. an array element A[J] as follows:

```
I := ADDRESSOF(A[J]) ;
ICL  (LDX, 2 I) ;
```

Execution of this code leaves the address of A[J] in X2 which might then be used as a modifier in accessing A[J] itself.

Chapter 2.

2. The 1900 Pascal compiler

The exact means of compiling, loading and running a 1900 Standard Pascal program depend on the particular computing installation and are defined in chapter 3. The present chapter describes the behaviour and output of the system in compiling and executing programs, and should be generally applicable.

2.1 Compilation

Compilation options are available to define the user's requirements in respect of source listing (see 2.1.1), run-time checking (2.1.2), addressing modes (2.1.3) and diagnostic facilities (2.4). These options are exercised by means of compiler directives (see section 2.5), or otherwise as provided by the installation (chapter 3).

2.1.1 The source program listing

The source program listing is a printed record of the program that was compiled. It consists of:-

(a) A heading showing the version of the compiler used, and the date and time of compilation.

(b) A line-by-line print-out of the source program including line numbers and object-code addresses.

Line numbers are successive integers starting from zero; they are thus compatible with the line-numbering conventions used elsewhere by 1900 software (e.g. the Editors).

The object-code address is an integer giving the location within the object program of the code generated from that line of source program. Where no such address exists, none is printed. These addresses are used in interpreting the error messages produced by the run-time monitor and the diagnostic system.

Errors detected during compilation are indicated by error message lines printed immediately after the source line in question. The exact position within this line to which the error message relates is shown by an upward-pointing arrow. The nature of the error is specified by numeric codes printed immediately after the arrow. These codes refer to the table of error messages given in section 2.3. Note, however, that the actual mistake may be distant from the point where the compiler detects the error.

(c) The listing is completed by a trailing summary indicating the success or otherwise of the compilation, the number of compilation errors reported, and descriptions of the checking, address-mode, and diagnostic options which have been selected.

There are compilation options making it possible to suppress the source listing, or to list specified parts of the program. However, lines containing errors are always printed, along with their corresponding error messages; as are the heading and the trailing summary.

2.1.2 The checking option

The compiler will incorporate into the object code, if requested, instructions to check against certain types of error happening during execution. These errors include:-

(a) The occurrence of overflow during integer or real arithmetic.

(b) Array subscripts, case indices, or values assigned to subrange variables being outside the range permitted.

These checks will in general produce some increase in the size and execution time of the object program, but their incorporation is nevertheless strongly recommended. Since the compiler exploits the information provided by subrange declarations to minimise the checking code produced, a rigorous use of subrange types may decrease the checking overhead substantially.

2.1.3 The address-mode options

The object program operates in a mode that depends on the range of addresses spanned by its instructions and working storage. Four cases can be distinguished:-

(a) If the complete storage requirement for the program, including both instructions and data, is less than 32K then code which operates in Compact Data Mode (CDM) may be used.

(b) If the data storage used during execution extends beyond 32K then Extended Data Mode (EDM) must be used.

(c) If the object program instructions lie entirely within 32K then Direct Branch Mode (DBM) may be used.

(d) If the instructions extend beyond 32K then Extended Branch Mode (EBM) must be used.

The modes required must be determined at the start of the compilation and must be the same for the whole of the program.

2.2 Execution

The only output from the execution of a 1900 Pascal program, apart from that written by the program itself, takes the form of termination messages. These will normally be made available to the user, in some form, by the operating system.

A 1900 Pascal program may terminate in one of four ways:-

(a) By reaching the final END of the main program block; in this case the final message is "OK".

(b) By performing a call on the HALT procedure; in this case the final message is that specified in the call of HALT. To avoid confusion with cases (a) and (c) this message should not begin with "OK" or "ERROR".

(c) By the detection of an execution error; in this case the final message takes the form

```
ERROR <error code> AT <object address>
```

The numeric error code again refers to the table given in section 2.3. The position within the source program at which the error occurred can be found by locating the given object address in the source program listing.

(d) By failing in a way which causes the Pascal run-time monitor to lose control. Such errors can usually be prevented by including run-time checks among the selected compilation options.

In cases (c) and (d) unfilled buffers will not be output to files open for writing. Consequently such files may be truncated and it may not be possible to re-read them satisfactorily.

After termination of the object program the 1900 Pascal "postmortem" system may be run to obtain diagnostic information (see section 2.4).

2.3 Error codes

2.3.1 Errors detected at compilation time only

1:	error in simple type
2:	symbol expected was an identifier
3:	
4:	symbol expected was ")"
5:	symbol expected was ":"
6:	unexpected symbol
7:	error in parameter list
8:	symbol expected was "OF"
9:	symbol expected was "("
10:	error in type
11:	symbol expected was "["
12:	symbol expected was "]"
13:	symbol expected was "END"
14:	symbol expected was ";"
15:	symbol expected was an integer constant
16:	symbol expected was "="
17:	symbol expected was "BEGIN"
18:	error in declaration part
19:	error in field-list
20:	symbol expected was ","
21:	misordered declarations ?
22:	only global variables may be initialised (non-Standard)
23:	
50:	error in constant
51:	symbol expected was ":="
52:	symbol expected was "THEN"
53:	symbol expected was "UNTIL"
54:	symbol expected was "DO"
55:	symbol expected was "TO" or "DOWNTO"
56:	symbol expected was "IF"
57:	
58:	error in factor
59:	error in variable
101:	identifier declared twice in the same block
102:	lower bound of subrange exceeds upper bound
103:	identifier is not of appropriate class
104:	identifier not declared
105:	a sign is not allowed here
106:	symbol expected was a number
107:	incompatible subrange types
108:	a file is not allowed here
109:	the type here must not be REAL

110:	a tagfield type must be ordinal
111:	incompatible with tagfield type
112:	an index type must not be REAL
113:	an index type must be ordinal
114:	a base type must not be REAL
115:	a base type must be ordinal
116:	error in type of standard procedure
117:	unsatisfied forward reference
118:	
119:	forward-declared: repetition of parameter list not allowed
120:	the type of a function must be ordinal, REAL or pointer
121:	file value parameters are not allowed
122:	forward-declared: repetition of result type not allowed
123:	missing result type in function declaration
124:	fixed-point format is allowed for REAL output only
125:	error in type of standard function parameter
126:	number of parameters does not agree with declaration
127:	
128:	result type of formal function conflicts with declaration
129:	types of operands conflict
130:	expression is not of SET type
131:	equality tests only are allowed for this type
132:	strict inclusion is not allowed (unfortunately)
133:	file comparison is not allowed
134:	illegal type of operand(s)
135:	type of operand must be "BOOLEAN"
136:	type of set element must be ordinal
137:	types of set elements are not compatible
138:	type of variable is not ARRAY etc
139:	subscript type is not compatible with declaration
140:	type of variable is not RECORD etc
141:	type of this variable must be pointer or FILE etc
142:	illegal parameter substitution
143:	illegal type of FOR-loop controlled variable
144:	illegal type of expression
145:	type conflict
146:	assignment of files is not allowed
147:	CASE label type incompatible with CASE selector
148:	subrange bounds must be ordinal
149:	subscript type must not be "INTEGER"
150:	assignment to a standard function not allowed
151:	assignment to a formal function is not allowed
152:	no such field in this record
153:	
154:	an actual VAR parameter must be a variable
155:	a FOR-loop controlled variable must not be formal
156:	multi-defined CASE label
157:	
158:	declaration of corresponding variant is missing
159:	REAL or string tagfields are not allowed

160:	
161:	multiple forward declaration
162:	parameter size must be constant
163:	
164:	standard procedure/function cannot be formal
165:	multiple definition of label
166:	multiple declaration of label
167:	undeclared label
168:	undefined label in block just ended
169:	error in base set
170:	formal procedure/function must have value parameters only
171:	
172:	undeclared external file
173:	a packed component cannot be passed as a VAR parameter
174:	forward procedure/function not found in block just ended
175:	subrange bounds cannot be REAL
176:	label not defined at correct level
177:	undeclared permanent file
178:	no INPUT or OUTPUT file has been specified
179:	
200:	
201:	error in REAL constant : digit expected
202:	a string constant must not cross line boundary
203:	INTEGER constant exceeds range
204:	8 or 9 in octal number
205:	strings of length zero are not allowed
206:	
250:	too many nested scopes (blocks/WITH statements)
251:	error in initialising global variables (non-Standard)
252:	too many forward references to procedure/function entries
253:	procedure too long
254:	
255:	too many errors on this source line
256:	
257:	
258:	
259:	expression too complicated
260:	expression or FOR/WITH nesting too complicated
261:	this should not occur : report compiler failure !
262:	
270:	code generated greater than 32K
296:	
297:	unimplemented compiler directive : ignored
298:	misplaced compiler directive : ignored
299:	invalid compiler directive : ignored

2.3.2 Errors detected at compilation time and at run time

300	
301:	no case provided for this value
302:	value outside expected range
303:	overflow or division by zero in INTEGER arithmetic
304:	overflow or division by zero in REAL arithmetic

2.3.3 Errors detected at run time only

305:	error in numeric input data
306:	error in output format
307:	insufficient storage available
308:	
309:	
310:	
311:	all logical-device unit numbers have been allocated
312:	failure to allot basic peripheral (CR/LP)
313:	failure to open magnetic peripheral (MT/DA)
314:	illegal attempt to write
315:	run off end of tape
316:	cannot extend scratch file/run off end of permanent file
317:	illegal attempt to read
318:	tape has non-Pascal format
319:	physical end of DA file
320:	illegal attempt to RESET a write-only file
321:	
322:	illegal attempt to REWRITE a read-only file
323:	
324:	tape format error (incorrect sentinel)
325:	physical end of tape
326:	reading after end-of-file
327:	writing before end-of-file
328:	printing off end of a line
329:	file specification cannot be read (on opening)
330:	file elements too large (>1023 words)

2.3.4 Errors detected at postmortem system run time only

401:	end of file
402:	too much output
403:	time up
404:	unallocated device
405:	illegal instruction (** see consultant **)

2.4 Diagnostic facilities

The Pascal diagnostic system supports source-language dumps, profiles, retrospective traces, and forward traces. The programmer is free to choose any combination of these.

After execution of the program, a termination message is printed which includes the CPU time used by the program, the maximum extents of the stack and heap (available only if a dump was requested), and, if the program failed during execution, a brief message describing the failure and its location within the program.

Following the termination message, the diagnostic information requested by the programmer is printed. Details of the diagnostic output are given in the following sections.

2.4.1 The postmortem dump

The postmortem dump, which is produced only if the program failed during execution, consists of a display of all blocks still active at the moment of termination.

For each block there is a heading containing the block's identifier and (where appropriate) the address of the point of call. Under this heading are displayed the values of all local variables and value parameters. As far as possible these are printed in source-language format, augmented by suitable notation for arrays and records.

Integer, real and character values are printed in their usual format. For types defined by enumeration of their values, including the Boolean type, each value is printed as an identifier. Any scalar value which is out of range is assumed to be undefined and is printed as "?"

A set value is printed in source-language format.

An array value is printed as a list of its elements enclosed in parentheses ("(", ")"). The elements are printed on separate lines if they are themselves structured, otherwise they are printed all on one line. At most 8 elements are printed, but these always include the first few elements and the last element.

As a special case, a packed-array-of-character value is always printed, in full, as a string.

A record value is printed as a list of its fields enclosed in angle brackets ("<", ">"). Any structured field is printed on a separate line, but consecutive unstructured fields are printed on a single line. Tag-fields are marked "VARIANT". Fields of variant-parts are printed, in number and format, as implied by the value(s) of the corresponding tag-field(s).

A pointer value is printed either as "NIL" or as an object-program address preceded by "@". (These are the only values not printed in source-language format. The representation chosen, however, allows pointer values to be compared for equality or inequality.) Data stored on the heap is not displayed.

In the case of a file variable, its current mode (reading or writing) and end-of-file status are indicated, and its current component is printed if it is defined.

2.4.2 The profile

The profile is an edited, automatically formatted, listing of the source program which displays the frequency of execution of each statement of the program.

The listing includes block headings and block bodies, but declarations and comments are not included. Indentation is used to emphasise the block structure and control structure of the program, and also to facilitate the interpretation of the profile, as described below.

To read off the frequency of execution of any statement, we locate the beginning of the statement in the profile listing. If an asterisk ("*") is present on the same line, to the left of the listing, then the desired frequency will be found to the left of the asterisk. Otherwise, the line will contain one or more upward-arrows ("↑"); we select the rightmost arrow, and follow the line of arrows vertically upwards until an asterisk is encountered; again, the desired frequency will be found to the left of the asterisk.

If the program terminated in failure, the profile must be interpreted with some caution. If the upward line of arrows being followed passes the failure point, or a procedure or function call leading to that failure point, then the frequency as obtained above will be one too high (possibly more, if the procedure or function call is recursive). As a warning, the location of the failure is marked prominently on the profile.

In addition to statements, the frequencies of execution of WHILE-expressions and UNTIL-clauses can be read independently off the profile. In the case of a WHILE-statement, for example, it is possible to read off, independently, the frequencies of execution of the WHILE-expression, of the repeated statement, and of the WHILE-statement as a whole.

2.4.3 The retrospective trace

The retrospective trace is a list of the last few "flow-units" executed up to the moment of termination of the program. (A "flow-unit" is a small piece of program, such as (a) a simple statement, or (b) an IF-, CASE-, FOR-, WHILE-, UNTIL-, or WITH-clause, or (c) simply "BEGIN","WHILE", or "REPEAT", indicating entry to a compound-, WHILE-, or REPEAT-statement.)

Each traced flow-unit is displayed in source-language form, accompanied by its object-program address and by a count N, indicating that this was the N-th execution of this particular flow-unit.

The number of flow-units in the retrospective trace is given by a run-time parameter RETROMAX, whose default value is 50; this value may be altered by means of a compiler directive or by suitable operating system commands before a run.

2.4.4 The forward trace

The forward trace is a list of <u>all</u> flow-units executed, with the exception of those executed while tracing was (temporarily) suspended. Each flow-unit is displayed in source-language form, and is accompanied by its address and count, as in the retrospective trace. Points where the trace was suspended are indicated.

The N-th execution of a flow-unit is traced only if TRACEMIN<=N<=TRACEMAX, where TRACEMIN and TRACEMAX are run-time parameters, with default values 1 and 2 respectively. So, when a flow-unit is executed whose count N does not satisfy this criterion, tracing is suspended (if not already suspended). Tracing is resumed when a flow-unit is executed whose count N does satisfy this criterion. The values of TRACEMIN and TRACEMAX may be specified by means of compiler directives, or by suitable operating system commands before each run.

By means of compiler directives, it is possible to restrict forward tracing to selected parts of the program only. Selective tracing is much more efficient than indiscriminate tracing. More importantly, however, the program can be viewed as a series of levels of abstraction, with each level embodied in a set of procedures; if only procedures in the higher level(s) are traced, no trace output at all will be obtained from the bodies of procedures in the lower level(s), which is nicely consistent with the view that the lower-level procedures merely implement the primitive operations of the higher level(s).

2.5 Compiler directives

Compiler directives appear within the source text as single lines distinguished by "%" in character position 1. Such lines are not considered to be part of the source program, in this respect behaving like comments. The same lexical conventions apply in directives as in Pascal texts. Any directives which are syntactically incorrect, or which are misplaced, will have no effect. A directive line may contain additional material to the right of the directive itself and clearly separated from it (e.g. by blanks). This will be treated as commentary.

Directives fall into three groups: those concerned with the handling of the source text, those concerned with the diagnostic facilities and those concerned with the kind of object code to be produced. In each group some directives (the initial directives) may appear only at the start of the source text, before the program proper. Others may appear at any point.

The environment in which the compiler is run may provide facilities, equivalent to certain directives, which allow options to be expressed without modification of the source text (see chapter 3).

2.5.1 Source-text handling directives

%LISTING = { TRUE | FALSE }

{ requests | suppresses } a source listing. The default is TRUE. This is an initial directive and must appear at the head of the source program (if needed).

%LISTING { ON | OFF }

may be used to restrict source listing to selected parts of the program, provided a source listing has been requested. Source text between any occurrence of %LISTING OFF and the next occurrence of %LISTING ON is not listed.

Regardless of any listing directives, a source line which is found to contain errors is always printed (together with the error indications).

%MARGIN = <integer>

specifies the number of characters examined by the compiler on each subsequent source record. The default is 80.

2.5.2 Diagnostic directives

%DUMP = { TRUE | FALSE }

{ requests | suppresses } a postmortem dump. The default is TRUE.

%PROFILE = { TRUE | FALSE }

{ requests | suppresses } an execution profile. The default is TRUE.

%RETRO = { TRUE | FALSE }

{ requests | suppresses } a retrospective trace. The default is FALSE.

%TRACE = { TRUE | FALSE }

{ requests | suppresses } a forward trace. The default is FALSE.

(The four directives above are all initial directives.)

%TRACE { ON | OFF }

may be used to restrict forward tracing to selected parts of the program, provided forward tracing has been requested. Any part of the program which lies between an occurrence of %TRACE OFF and a subsequent occurrence of %TRACE ON will <u>not</u> be traced.

%RETROMAX = <integer>

specifies the number of flow units to appear in the retrospective trace. The default is 50.

%TRACEMIN = <integer>
%TRACEMAX = <integer>

specify the cutoff counts for forward tracing: the Nth execution of a flow-unit will be traced only if:-

TRACEMIN <= N <= TRACEMAX

The defaults are 1 and 2 respectively.

2.5.3 Object code directives

%CHECKS = { TRUE | FALSE }

{ requests | suppresses } the inclusion of object code to perform run-time checking for range and overflow errors. The default is TRUE.

%CDM = { TRUE | FALSE }

{ requests | suppresses } the generation of code to run in CDM (Compact Data Mode) which limits the total program size to 32K words or less. The default is FALSE.

%EBM = { TRUE | FALSE }

{ requests | suppresses } the generation of code to run in EBM (Extended Branch Mode) which must be used if the program instructions exceed 32K words. The default is FALSE.

(The three directives above are initial directives).

Example

A program P is required to produce a dump if it fails, but no profile or retrospective trace. Forward tracing is required for a procedure Q alone, the cutoff counts being 1 and 6.

```
%PROFILE = FALSE
%TRACE = TRUE
%TRACEMAX = 6
%TRACE OFF
PROGRAM P;
...
PROCEDURE Q;
  ...
%TRACE ON
   BEGIN (* Q *)
      ...
   END (* Q *) ;
%TRACE OFF
 ...
BEGIN (* P *)
   ...
END .
```

2.6 The source library mechanism

A 1900 Pascal enhancement is available which enables compilers running under George 3 or 4 operating systems to incorporate procedures or functions from a source library during compilation. The procedure or functions are held as normal text in file store files, either within the user's own file area (a private library) or in a system defined area which represents a public library. The library may thus be created and maintained using the normal text filing and editing facilities of the George environment.

2.6.1 Retrieving a library procedure or function

To request the incorporation of a library procedure or function during compilation the user includes a procedure or function <u>retrieval command</u> at the appropriate point in a procedure or function declaration part of his program. The syntax of a procedure or function declaration is extended as follows

```
<procedure or function declaration> ::=
     <procedure declaration> | <function declaration> | <retrieval command>
<retrieval command> ::=
     PROCEDURE <identifier> <retrieval specification> |
     FUNCTION  <identifier> <retrieval specification>
```

The retrieval specification determines the file from which the procedure or function is to be retrieved, whether listing is to continue during compilation of the procedure, and any adjustment of the non-local identifier scope necessary for its proper compilation.

```
<retrieval specification> ::= <library file specification>
                              <listing specification>
                              <interface scope specification>
```

The <u>library file specification</u> takes the following form

```
<library file specification> ::= <filename specification>IN<area specification>
<filename specification> ::= <empty> | = <identifier>
<area specification> ::=  LIBRARY | PRIVATE LIBRARY
```

If no filename specification is given the required file is assumed to have the same name as the procedure identifier specified by the retrieval command, otherwise it is the identifier following = . Note that since 1900 Pascal identifiers are limited to 8 significant characters the names used for library filestore files must also be limited to 8 characters or less, in the form of a valid Pascal identifier.

If PRIVATE appears in the area specification the file specified is sought in the user's own file store area, otherwise it is sought in a system-designated public area.

The <u>listing specification</u> takes the form

```
<listing specification> ::= <empty> | , LIST
```

If , LIST is present listing continues during compilation of the retrieved library procedure but with line numbers restarting from zero, to indicate the source line position within the library file. (When compilation of the normal program text following the retrieval command is resumed, the previous sequence of line numbers is also resumed.) If the listing specification is empty listing is suppressed during compilation of the library procedure.

The meaning of a procedure or function often depends on identifiers denoting types or constants which are assumed to be non-local to the procedure or function block. It is often inconvenient,or impossible, to ensure that the required identifiers are defined in the scope in which a procedure or function retrieval command occurs. Instead the user may define such identifiers in an <u>interface scope specification</u> within the retrieval command itself. Identifiers defined in this way are available as non-locals during the compilation of the retrieved procedure or function, but have no effect on the scope in which the retrieval command occurs.

```
<interface scope specification> ::=
     WITH <interface constants> |
     WITH <interface types> |
     WITH <interface constants> ; <interface types>

<interface constants> ::= CONST <constant definition>
                                { ; <constant definition> }

<interface types> ::= TYPE <type definition>
                           { ; <type definition> }
```

Examples of retrieval commands:

```
PROCEDURE SORT IN LIBRARY ;

PROCEDURE SORT = MYSORT IN PRIVATE LIBRARY, LIST ;

PROCEDURE SORT IN LIBRARY
    WITH CONST N = 100 ;

PROCEDURE SORT IN LIBRARY
    WITH CONST N = 100 ;
         TYPE TABLE = ARRAY [1..N] OF REAL ;
              ELEMENT = REAL ;
```

2.6.2 Storing library procedures and functions

The contents of the filestore file retrieved by a retrieval command must be a 1900 Pascal procedure or function declaration terminated by a semi-colon in the usual way. The identifier appearing in the procedure or function heading is irrelevant since the procedure or function compiled will be identified by the name given in the retrieval command. To avoid confusion it is recommended that the name appearing in the filed procedure or function heading should coincide with the filename by which it is retrieved. A recursive procedure or function must always be retrieved with the name by which it recursively calls itself.

Library procedures and functions may contain retrieval commands for other library procedures or functions. In general retrievals may be nested to any depth, provided a recursive retrieval cycle is not set up.

2.6.3 Generalised procedures and functions

The fact that the meaning of a procedure or function depends on the non-local type and constant identifiers used within it enables quite general procedures and functions to be stored in source libraries. The effect of the procedure in any particular compilation will be determined by the definition given to these identifiers in the scope in which the retrieval command occurs, or by the retrieval command itself. To facilitate the programming of such general procedures and functions one further extension is made to 1900 Pascal by the source library enhancement. The minimum and maximum values allowed by a simple type T may be denoted in an expression as T.MIN and T.MAX respectively. The syntax of expression is thus extended as follows

```
<unsigned constant> ::= <unsigned number> | <string> |
                        <constant identifier> | <type limit> | NIL

<type limit> ::= <type identifier>.MIN | <type identifier>.MAX
```

The type denoted by the type identifier may be any simple type other than REAL.

2.6.4 Error Codes

281: File not found

282: Error in retrieval specification

283: Type here must not be real

284: Symbol expected was "MIN" or "MAX"

285: Symbol expected was "."

Chapter 3.

3. Using 1900 Pascal under GEORGE 3/4

3.1 The PASCAL command

The PASCAL command is a GEORGE macro which controls the compilation and execution of 1900 Standard Pascal programs. A variety of parameters can be supplied, but all of them are optional and have simple defaults. Each parameter is identified by a keyword so they can appear in any order. Parameters are separated by commas, leading and trailing spaces being ignored. All workfiles created in the macro are erased before it exits.

TEXT=<text>

The program text is compiled from file <text>, or from the job deck by default. Several parameters of this form may be given, in which case the source text is read from each of the files named, in turn from left to right.

LIST=<listing>

The compilation listing is sent to the file <listing>, or to a workfile by default. The listing is printed automatically if a workfile is used, otherwise the user must make his own arrangements to print it.

OPTIONS=(<O1>, ... ,<On>) or OPTION=(<O1>)

The option(s) <Oi> are used to over-ride the compiler's default settings at the start of the compilation. See section 3.3.

NORUN

If this parameter is present the object program is not executed.

SAVE=<save>

The object program is saved in the file <save>, or not saved by default.

TIME=<time>

The object program is given a CPU time limit (for this run only) of <time>, or 10SECS by default.

RUNOPTIONS=(<O1>, ... , <On>) or RUNOPTION=(<O1>)

The option(s) <Oi> are used to over-ride (for this run only) the trace limits set when the program was compiled. See section 3.3.

DUMP=(<area>)

Areas <area> of the object program are dumped to the monitoring file (see section 3.6), by default nothing is dumped. This dump takes place after the execution of the object program, so the parameter has no effect if NORUN is also specified.

PREDUMP=(<area>)

This parameter acts like DUMP, but is obeyed before the execution of the program. Hence it is unaffected by NORUN.

INPUT=<input>

The data comprising standard file "INPUT" is taken from GEORGE file <input>, or from the job deck by default. Several parameters of this form may be given, in which case the data is read from each of the files named, in turn from left to right.

OUTPUT=<output>

The data comprising standard file "OUTPUT" is sent to the GEORGE file <output>, or to a workfile by default. The file is printed automatically if a workfile is used, otherwise the user must make his own arrangements to print it.

FILES=(<F1>, ... ,<Fn>) or FILE=(<F1>)

A parameter of this form is needed if any permanent FILE variables are declared. See section 3.4 for details.

DEVICES=(<Dd1>, ... , <Ddn>) or DEVICE=(<Dd1>)

A parameter of this form is used to change (for this run only) the devices associated with some or all of the permanent FILE variables. See section 3.4 for details.

BINARY=<binary>

The binary program in file <binary> is loaded and run. It must be a Pascal object program produced by the SAVE parameter in another compilation. If the BINARY parameter is present all parameters to do with compiling a source text are ignored. Any parameters relating to the object program may be used along with BINARY.

TOKEN=<token> or TABLE=<table>

These parameters are used in conjunction with the source-language diagnostics. See section 3.5 for more details.

CORE=<core>

The object program is allowed to use a maximum (for this run only) of <core> words of store, or 49152 by default.

FAIL=<n>

If the compilation fails for any reason and this parameter is present the PASCAL macro performs a jump to the JCL label <n>FAIL. This allows subsequent commands to be conditional on the outcome of the compilation.

TRACE=(<level>)

The JCL tracing level is set to <level> within the PASCAL macro. The default level is "FULLBUT, COMMANDS, COMMENT".

The following parameters are chiefly of value in maintaining the Pascal system itself, but may occasionally find some more general applications.

COMPILER=<comf> or MONITOR=<monf> or POSTMORTEM=<pmdf>

The file named in the parameter is loaded and run during the appropriate phase of the macro, in place of the usual component.

OBJECT=<ocf>

The object code generated by the compiler is copied to the file <ocf>. This cannot be run as a complete program until it is bound to a run-time monitor, e.g. by means of the BINDPASCAL command.

N.B.

The file parameters denoted <text>, <input>, <binary>, <comf>, <monf> and <pmdf> above must already exist when they are used in a (BIND)PASCAL command. Parameters <listing>, <output>, <save>, <ocf>, <token> and <table> will be created if they do not already exist. If any of the latter four already exist they must be direct-access files.

A qualifier consisting of the words "READ", "WRITE" or "APPEND" in parentheses may be placed after one of these filenames. This is mandatory for direct-access files and must be "READ" or "WRITE". For basic-peripheral files only "APPEND" is relevant.

3.2 The BINDPASCAL command

The BINDPASCAL macro accepts the same parameters as the PASCAL command, with the exception of those relating to the compilation of a source text. An object code file and a run-time monitor must be specified: an executable program is created by binding these together. Any other actions of the macro are determined by such additional parameters as are given, these being interpreted in the same way as by the PASCAL macro.

3.3 The OPTION and RUNOPTION parameters

The OPTION parameter allows the compiler's initial settings for the various compilation options to be varied without the need to insert directives at the start of the source program. The RUNOPTION parameter similarly allows some control over the run-time facilities which are used in any one execution of an object program. For an understanding of the effects of these options, see sections 2.4 and 2.5.

```
<Oi> ::= <compilation-only option>
       | <execution-only option>

<compilation-only option> ::= CHECKS | NOCHECKS
                            | DUMP | NODUMP
                            | PROFILE | NOPROFILE
                            | RETRO | NORETRO
                            | TRACE | NOTRACE
                            | <address mode>

<address mode> ::= EDM | CDM | DBM | EBM

<execution-only option> ::= RETROMAX=<integer>
                          | TRACEMIN=<integer>
                          | TRACEMAX=<integer>
```

In the absence of a contrary compiler directive or OPTION parameter, programs are compiled with the option settings CHECKS, DUMP, PROFILE, EDM, CDM, RETROMAX=50, TRACEMIN=1 and TRACEMAX=2. In the absence of a RUNOPTION parameter, programs are executed with the option settings applying when they were compiled.

3.4 The FILE and DEVICE parameters

A Pascal program may declare permanent FILE variables to which external files are to be connected. Under GEORGE 3/4 this is done in two distinct stages.

Firstly, a device type and logical unit number are associated with each of the permanent FILE variables. Default associations are set up by the compiler (see section 1.6). If it is necessary to change any of these for a particular execution of the program this can be specified by the DEVICE parameter. For each file variable to be bound to a new logical device there must be given (a) its position in the program parameter list [counting from zero], (b) the device type and unit number to be used and (c) optionally, an indication of whether the file is to be opened for read-only access. The syntax is as follows.

```
<Ddi> ::= (<file parameter no.>,<device type>,<device no.><flag>)

<file parameter no.> ::= <integer>

<device type> ::= *LP | *CR | *MT | *DA

<device no.> ::= <integer>

<flag> ::= , READ | <empty>
```

Secondly, the file which the logical device will access must be specified before each execution. This may be an entrant in the filestore, data held in the job deck itself, or results to be printed in the job monitoring output. The FILE parameter connects the logical devices of a program with the files to be accessed by giving a series of equations between the logical device names and the corresponding actual files.

```
<Fi> ::= <logical device>=<actual file>

<logical device> ::= <device type><device no.>

<actual file> ::= <GEORGE 3/4 entrant description>
```

The standard file variable INPUT on logical device *CR0 may be specified by an INPUT parameter instead of a FILE parameter. This makes available a data concatenation facility which is not possible if *CR0 is specified by the FILE parameter. Similarly the standard file variable OUTPUT on logical device *LP0 can be specified by the OUTPUT parameter. When the FILE parameter and the { INPUT | OUTPUT } parameter conflict, the FILE parameter takes precedence.

Note that the actual file description must include a (WRITE) qualifier if the corresponding logical device is of direct-access type and has not been limited to read-only opening by the DEVICE parameter. (This applies even if the program never writes to the file.)

3.5 Diagnostics from saved programs

The compiler writes two files containing information for use by the postmortem diagnostic system - the "token" file, which contains a condensed version of the source text and the "table" file, which contains a summarised symbol table. When the PASCAL macro is called to compile and run a program, workfiles are used to pass this data to the postmortem program. However, if it is desired to obtain diagnostics from a binary program created by the SAVE parameter, then the contents of these files must be retained. The TOKEN and TABLE parameters are provided to allow the user to nominate permanent files in place of the default workfiles. The "table" file need be kept only if symbolic dumps are wanted, the "token" file is needed for profiles, retrospective tracing and forward tracing.

3.6 Machine code dumps

The (PRE)DUMP parameter is provided to enable the object program to be examined. PREDUMP takes effect before the program is run and DUMP afterwards. Otherwise they are identical. They may be used together for "before and after" comparison. When a single area is to be dumped the parameter has the form:-

```
(PRE)DUMP=(<area>)
```

and several areas can be dumped by the form:-

```
(PRE)DUMP=(<area 1>, ... , <area N>)
```

where:-

```
<area i> ::= <first>(<number>) | (<first>,<last>)
```

Here <first> and <last> are addresses and <number> is the size of the area to be dumped.

Dumps are printed in the job monitoring file and they should be used sparingly. The rigorous run-time security and the postmortem symbolic diagnostics of 1900 Standard Pascal make machine code dumps unnecessary in all but the very direst of circumstances. (See section 3.3).

3.7 Use of PASCAL command, simple examples

```
PASCAL
```

Compile and, if no compilation errors, run a PASCAL program. Source and data are read from cards, listing and results are sent to the lineprinter. Default OPTIONS apply i.e. checks are incorporated into the object code, a postmortem dump, profile and traces are provided if the program fails.

```
PASCAL TEXT=FILE1,INPUT=FILE2,SAVE=BINFILE
```

The source in FILE1 is compiled. If there are no compilation errors, the binary program is saved in BINFILE and then run, data being read from FILE2. Listing and results are sent to the lineprinter. Default OPTIONS apply.

```
PASCAL BINARY=BINFILE,INPUT=FILE3,TIME=20,CORE=20000
```

The binary program in BINFILE is loaded and run. It is allowed 20000 words of core (instead of default 49152) and 20 secs of mill time (default 10). Data is read from FILE3 and the results sent to the lineprinter. The OPTIONS invoked at compile time now apply.

The 1900 PASCAL system is supplied as a binary compiler program, #PASQ, and a binary post-mortem generated #POST.

1. Compiling a PASCAL program

#PASQ should be loaded and the compilation options chosen should be indicated by setting switch word 30 as follows

switch 23

on	-	suppresses runtime error checks
off	-	generates runtime error checks

switch 22

on	-	generates compact data mode program (CDM or 15AM)
off	-	generates extended data mode program (EDM or 22AM)

switch 21

on	-	generates extended branch mode program (EBM)
off	-	generates direct branch mode program (DBM)

switch 20

on	-	suppresses runtime symbolic dumps
off	-	allows runtime symbolic dumps

switch 19

on	-	suppresses execution profiling
off	-	allows execution profiling

switch 18

on	-	allows retrospective tracing
off	-	suppresses retrospective tracing

switch 17

on	-	allows forward tracing
off	-	suppresses forward tracing

switch 16

on	-	suppresses source listing
off	-	allows source listing

These settings have been chosen so that an all-zero switch word gives what is thought to be the most desirable combination of options, i.e. CHECKS, DUMP and PROFILE.

#PASQ should be entered by :-

```
GO #PASQ 20
```

Since #PASQ is itself a PASCAL program with heading :-

```
program PASQ (INPUT,OUTPUT,CODEFILE,OBJECTFILE,TOKENFILE)
```

it will expect to read the source text from *CRO, to output the listing to *LPO and to output the object program to CODEFILE. The object table is output to OBJECTFILE and the stripped source text to TOKENFILE. OBJECTFILE and TOKENFILE are used by the postmortem diagnostic system. They must be supplied, even if all diagnostic options are suppressed. CODEFILE, OBJECTFILE and TOKENFILE are all direct-access files, opened as *DAO, *DA1 and *DA2.

Compilation should terminate in one of two ways

(i) #PASQ HALTED := CC

This indicates that compilation has been completed successfully and that a loadable object code program has been written to CODEFILE.

(ii) #PASQ HALTED :- CE

This indicates that compilation has been completed with one or more errors having been found in the source program. A loadable object code program is not produced in this case.

Any other termination represents a failure of compilation for one of two reasons, either

(a) insufficient working storage is available for compilation to continue, or

(b) a previously undetected error in the compiler has occurred.

Case (a) will normally be indicated by a PASCAL error halt of the form

```
#PASQ HALTED :- ERROR 307 AT ....
```

In this case compilation may be re-attempted with an increased storage limit. In all other cases the compiler failure should be reported to the authors with a listing of the source program which caused it.

2. Loading a PASCAL program

To produce a loaded executable program from the code file generated by compilation the compiler #PASQ is (re-)entered as follows

```
GO #PASQ 21
```

This will open the direct access file CODEFILE (as DAO), read down the object code to form a loaded executable program, change its execution mode to that requested for the object program, change its name to the first four characters of that used in the source program heading, and halt :

```
#nnnn HALTED :- LD
```

The program #nnnn is then suitable for execution as described in 3 below, or for saving as a standard ICL format binary program.

If the code file has a name other than CODEFILE, the name may be changed in the loader by altering words

786 ,787 ,788 to the new name.

Loading may halt prematurely for one of three reasons

(i) #PASQ HALTED :- NF

in this case the loader has failed to open the required direct access file CODEFILE

(ii) #PASQ HALTED :- IC

in this case insufficient core is available for the object program to be loaded. If additional core can be made available loading can be restarted by GO #PASQ

(iii) #PASQ HALTED :- UP

in this case an unclearable parity has occurred in reading the object code from the code file.

3. Executing a PASCAL program

When loaded a binary program produced as described in 1 and 2 should be entered by

```
GΘ #nnnn 20
```

The executing program may halt with one of the following messages

(i) #nnnn HALTED :- OK

in this case execution has terminated by reaching the end of the main program block.

(ii) #nnnn HALTED :- xxxxxx...

in this case execution has terminated by executing a call to the procedure halt of the form

```
halt ('xxxxxx...')
```

(iii) #nnnn HALTED :- ERRΘR eee AT ppppp

in this case execution has terminated on the detection of error eee at program location ppppp, as explained in the 1900 PASCAL user guide.

4. Obtaining post-mortem diagnostic information

If diagnostic options have been specified during its compilation, execution of a binary 1900 Pascal program will open an additional direct access file PASQDIAGFILE as DA63, in which diagnostic information is accumulated during execution.

Failure to open this file is indicated by a halt

```
#nnnn HALTED :- ND
```

When execution terminates as described in section 3 the HALTED message will be preceded by a DISPLAY message of five characters indicating the postmortem diagnostic options in effect and the status of the program on termination. These five characters are independently significant, as follows:-

(1) D if a post-mortem dump is available, otherwise X

(2) P if a profile is available, otherwise X

(3) R if retrospective tracing is available, otherwise X

(4) T if forward tracing is available, otherwise X

(5) E if a runtime error has been detected, otherwise X

If execution is interrupted for any other reason the proper writing of the diagnostics file, and an appropriate halt message, can be obtained by re-entering the object program by

```
                    GΘ #nnnn 29
```

after altering register 3 to the most appropriate of the error codes 401 .. 405 given in section 2.3 of the User Guide.

To obtain the post-mortem information specified the post-mortem generator #POST should be loaded and entered by

```
                    GΘ #PΘST 20
```

The generator program will open the OBJECTFILE and TOKENFILE written at compile time, and the diagnostic file PASQDIAGFILE written at runtime, as *DA0, *DA1 and *DA2 respectively. The diagnostic information required will then be output on lineprinter *LP0 as described in the User Guide.

5. Core allocation in the 1900 PASCAL system

Both #PASQ itself and the object programs which it generates are designed to operate in a controlled storage environment. In the binary version of #PASQ supplied and in the binary programs produced by it, the core store request word of the request block is set to the minimum required to accommodate the static binary program. When entered the binary program automatically attempts to extend its core store allocation to a standard working limit (for compact data mode programs this limit is 32K, otherwise it is 256K). The program then proceeds as far as possible with the store allocation so obtained, whether it is that requested or less.

The standard working limit may be reset before entering the binary program by altering the contents of word /46 to the new limit value, e.g. by a directive to EXEC

```
                    AL #nnnn     limit
```

This applies equally to #PASQ and its generated programs.

6. File mapping

As described in the 1900 PASCAL User Guide, the 1900 PASCAL system adopts a standard mapping for the external files of a PASCAL program onto corresponding physical devices, as follows

(i) the standard file INPUT is implemented as a card reader CR0

(ii) the standard file OUTPUT is implemented as a line printer LP0

(iii) all other external files are implemented as direct access files with the same name as the file variable, opened in read/write mode as DAn where the device unit numbers n are allocated as 0,1,2,.. in the order of occurrence of the file names in the program heading. (These files must already have been created).

For most users in, say, a George filestore environment a simple re-mapping of these physical files at George level will meet all their needs. Some users may however require more precise control over the mapping. This is made possible within the 1900 PASCAL system in the following way.

For each external file named in the program heading the compiler generates a <u>device record</u> in the first available locations of the object program, i.e. those starting at the object address listed on the program heading line. Each device record consists of 6 words whose contents determine the corresponding physical device used, as follows

word 0   determines the opening mode as follows

0 = read only
1 = write only
2 = read/write

word 1   determines the device unit number to be used 0,1,2, etc.

word 2   determines the device type to be used

0 = LP
1 = CR
2 = MT
3 = DA

(these are the only device types allowed)

words 3,4,5   determine the filename to be used when appropriate.

By altering any or all of these device record locations before execution commences, the physical devices used by the program can be controlled. Card readers and line printers can only be used for external text files.

7. Representation of files

The representation of files used on magnetic tape or direct access media is peculiar to the 1900 PASCAL system and not designed to be compatible with other 1900 software. Details of the representation used can be obtained from the listings of the corresponding support routines within the loaders. Text files which are to be processed by non-PASCAL software must therefore be manipulated as card-reader or line-printer files.

An input card-reader file is assumed to be terminated by a card with asterisks (****) punched in columns 1-4. Variation of this end-of-file marker can be achieved by altering location 147 of the object program to hold the four-character marker required.

8. The source library enhancement

The source library enhancement is supplied as a George edit for the original Pascal compiler source. The edited compiler source should be compiled (as an EBM program) to produce a binary compiler which implements the library mechanism.

The resultant compiler has the same operational characteristics as the original #PASQ compiler except that the source to be compiled is read using a George file reader *FR0, which should therefore be assigned to the appropriate input file before the compiler is entered. Thereafter the compiler dynamically issues George commands to assign further file readers *FR1, *FR2, etc. according to the depth of nesting of retrieval commands which it encounters.

The compiler assumes that 'public' library files are held under a pre-determined user identity code. The code used is determined by the string assignment which is the first action of the procedure ATTACHFILE inserted by the edit supplied. The code may therefore be varied by first editing this string assignment. All public library files must have traps set to allow read access to all users entitled to retrieve them.